Fleet Optimizer Overall Architecture Description

The main output of this application written in Python - Django is a simulation which results in an optimal solution with respect to number of trips made to cover a tonnage (work) demand assigned to each main pit route in a mine.

The abstract model is comprised of a hierarchy of classes corresponding to each of its elements; the destinations in the mine such as the garage, shovels and loaders, the routes that connect each destination or point, the trucks that transit these routes, etc. Each of these elements has its own

properties, for example the trucks have a specific nominal capacity and the points or destinations will only withstand a specific amount of trucks in queue.

The process evolves in discrete time where each step represents a time segment which is basically the cycle time calculated from the origin data (data source).

In each time segment there are a series of actions that need to be considered. Each action incurs a cost and result in a change of state of the problem. More concretely, the actions correspond to any truck assignment possible.

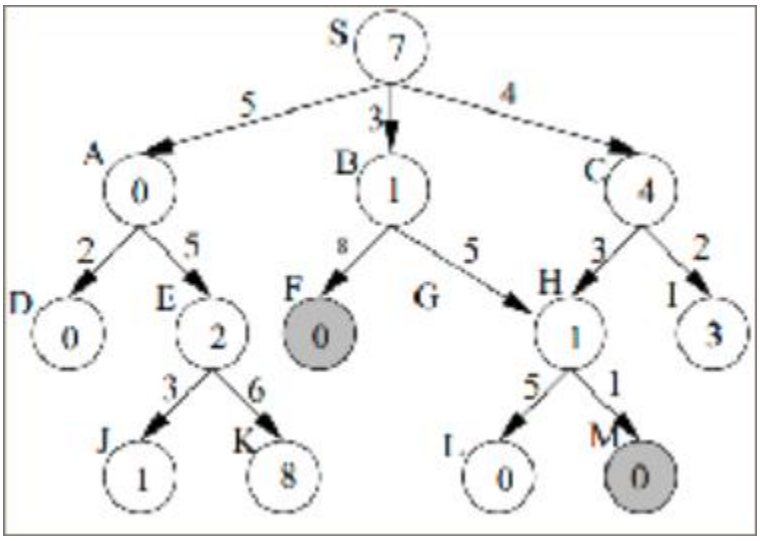
The current app is set up to follow my paper´s madel where we have a system of 7 destinations and 21 trucks each with 100 tonne capacity (based on a real mine where I used to work).

In a second step of the simulation the trucks begin traveling the routes towards the different destinations back and forth. If the truck is carrying a load the resulting action satisfies partially the demand on that route and the truck remains on the same destination where it started during that time segment. However if the truck gets assigned to a new route the demand satisfied previously does not decrement.

Each of these actions incurs a cost, which is a full trip, and this is the amount we want to minimize choosing the actions that satisfy all the demand of the different routes with the truck capacity assigned.

If we are to visualize the process as a decision tree, we consider that each node corresponds to a possible state of the problem. The first time segment represents initial state which is the root of the

decision tree. Each possible action represents a brach off from the actual node resulting in a new state.



As the process evolves the decisión tree eventually arrives at an end. All the possible endings are the leaves of the tree and in the special case of the Fleet Optimizer these leaves represent states where the demand on each route has been satisfied or the time segments available in the shift have been exhausted.

Not all ending states are aceptable, only the ones that have satisfied the demand on the routes and have returned to the garage in order to comply with the fleet conservation constraint (see paper for more detail on that).

Solution Search

Any algorithm that searches for the best solution has to travel the decision tree until finding the best solution. An exhaustive search is not feasible given that the possibilities explode exponentially.

A more efficient search prioritizes resulting elements of each action that have a minor cost. In this case the actions that minimize the number of trips. Uniform Cost Search (UCS) is an efficient algorithm that considers the sequences of actions using a priority queue utilizing the cost of each route in the tree as the main criteria for the search.

A* is an extension of UCS that uses the cost plus a heuristic to estimate the final cost in order to reach an optimal solution. The heuristic must be expressed in terms compatible with the cost and should not over estimate the optimal cost.

Code Element: Core_Search

Core_Search is a python package that takes fleet data (trucks) and parameters pertaining to the optimization problem and executes the simulation.

Core_Search takes the mine configuration like a list of connections which are represented as Python tuples (immutable list), whose elements are assets in the mine that communicate (i.e. the different destinations in the mine). For example:

```
# First build the locations
shovel1 = Location("S1", 2)
shovel2 = Location("S2", 2)
loader1 = Location("L1", 2)
loader2 = Location("L2", 2)
waste_dump = Location("W", 2)
crusher = Location("C", 2)
garage = Location("garage", 21)

# Build the mine configuration
connections = [
    (garage, loader1),
    (garage, loader2),
    (loader1, garage),
    (loader2, garage),
    (waste_dump, garage),
    (garage, shovel1),
    (garage, shovel2),
    (waste_dump, loader1),
    (loader1, waste_dump),
    (waste_dump, shovel1),
    (shovel1, waste_dump),
    (waste_dump, shovel2),
    (shovel2, waste_dump),
    (crusher, shovel1),
    (shovel1, crusher),
    (crusher, shovel2),
    (shovel2, crusher),
    (crusher, loader2),
    (loader2, crusher),
    (loader1, crusher)
```

The truck fleet is defined as a sequence of **Truck** type objects:

```
# Generate the trucks
trucks = [Truck("truck_%i" % i, c) for i, c in zip(range(1, num_trucks+1), it.cycle([100]))]
```

The tonnage demand on each route is defined as a dictionary where the route is the key and the tonnage demand is the value:

```
demands = OrderedDict(
[((shovel1, crusher), 8000),
    ((shovel2, crusher), 1200),
    ((loader1, crusher), 4000),
    ((shovel1, waste_dump), 1600),
    ((shovel2, waste_dump), 2000),
    ((loader1, waste_dump), 1000)]
)
```

The elements of the simulation are contained within a State object:

```
# Create the initial state
initial_state = FleetState(config, trucks, demands, num_segments)
```

The initial state represents the mine shift start where all trucks are located in the main garage and all routes have not been travelled. Core_Search implements the A* algorithm which executes the simulator looking for the state that covers all route demands minimizing the number of trips required:

```
searcher = AStar(initial_state, heuristic, listener)
solution = searcher.solve()
```

If the simulator finds a solutions, it reaches the final state of simulation which can be used to reconstruct the sequence of actions used to reach such final state. If the simulation does not find a solution it returns to None state.

A detailed example of how to run the simulator can be found in the core_search/run.py file.

Code element LeanUI:

The lean_ui module is actually a portal developed in Django which imitates the script run.py and is an example of how to integrate core_search inside a separate information system. A rudimentary (no flashy yet) web page can be found at http://127.0.0.1:8000/fleet

For the project to run it is important that the core_search module is accessible from PYTHONPATH. The dependencies pertaining to this module are found in the “requirements.txt” file within lean_ui.